

AD-A235 831 TATION PAGE

Form Approved
OPM No. 0704-0188

Pub
nee
Has
Mat



Our per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington after Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE		3. REPORT TYPE AND DATES COVERED Final: 09 Sept 90 to 03 Mar 92=3	
4. TITLE AND SUBTITLE Meridian Software Systems, Inc., Meridian Ada, Version 4.1, IBM PS/2 30 (Host & Target), 900909W1.11035				5. FUNDING NUMBERS	
6. AUTHOR(S) Wright-Patterson AFB, Dayton, OH USA					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Ada Validation Facility, Language Control Facility ASD/SCCL Bldg. 676, Rm 135 Wright-Patterson AFB Dayton, OH 45433				8. PERFORMING ORGANIZATION REPORT NUMBER AVF-VSR-395.0491	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Ada Joint Program Office United States Department of Defense Pentagon, Rm 3E114 Washington, D.C. 20301-3081				10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) Meridian Software Systems, Inc., Meridian Ada, Version 4.1, Wright-Patterson AFB, IBM PS/2 30 IBM PC-DOS 3.30 (Host & Target), ACVC 1.11.					
14. SUBJECT TERMS Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.				15. NUMBER OF PAGES	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED		20. LIMITATION OF ABSTRACT	

DTIC
ELECTE
MAY 30 1991
S E D

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 09 September 1990.

Compiler Name and Version: Meridian Ada, Version 4.1

Host Computer System: IBM PS/2 30
IBM PC-DOS 3.30


Target Computer System: IBM PS/2 30
IBM PC-DOS 3.30

Customer Agreement Number: 90-07-23-MSS

See section 3.1 for any additional information about the testing environment.

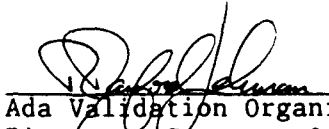
As a result of this validation effort, Validation Certificate 900909W1.11035 is awarded to Meridian Software Systems, Inc. This certificate expires on 1 March 1993.


This report has been reviewed and is approved.


Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503



Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	


Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311


Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

91-00775



91 5 29 111

AVF Control Number: AVF-VSR-395.0491
8 April 1991
90-07-23-MSS

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 900909W1.11035
Meridian Software Systems, Inc.
Meridian Ada, Version 4.1
IBM PS/2 30 => IBM PS/2 30

Prepared By:
Ada Validation Facility
ASD/SCSL
Wright-Patterson AFB OH 45433-6503

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 09 September 1990.

Compiler Name and Version: Meridian Ada, Version 4.1

Host Computer System: IBM PS/2 30
IBM PC-DOS 3.30


Target Computer System: IBM PS/2 30
IBM PC-DOS 3.30

Customer Agreement Number: 90-07-23-MSS


See section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 900909W1.11035 is awarded to Meridian Software Systems, Inc. This certificate expires on 1 March 1993.

This report has been reviewed and is approved.



Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503



for Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

Customer: Meridian Software Systems, Inc.

Ada Validation Facility: ASD/SCEL, Wright-Patterson AFB OH 45433-6503

ACVC Version: 1.11

Ada Implementation:

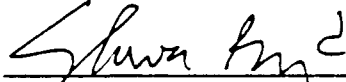
Compiler Name and Version: Meridian Ada, Version 4.1

Host Computer System: IBM PS/2 Model 30 (with Floating Point Co-Processor)
IBM PC-DOS 3.30

Target Computer System: IBM PS/2 Model 30 (with Floating Point Co-Processor)
IBM PC-DOS 3.30

Customer's Declaration

I, the undersigned, representing Meridian Software Systems, Inc., declare that Meridian Software Systems, Inc. has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation listed in this declaration. I declare that Meridian Software Systems, Inc. is the owner of the above implementation and the certificates shall be awarded in the name of the owner's corporate name.



Date: 3 Aug 1990

Stowe Boyd, Vice President of Research and Development
Meridian Software Systems, Inc.
10 Pasteur Street
Irvine, CA 92718

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	USE OF THIS VALIDATION SUMMARY REPORT	1-1
1.2	REFERENCES	1-2
1.3	ACVC TEST CLASSES	1-2
1.4	DEFINITION OF TERMS	1-3
CHAPTER 2	IMPLEMENTATION DEPENDENCIES	
2.1	WITHDRAWN TESTS	2-1
2.2	INAPPLICABLE TESTS	2-1
2.3	TEST MODIFICATIONS	2-4
CHAPTER 3	PROCESSING INFORMATION	
3.1	TESTING ENVIRONMENT	3-1
3.2	SUMMARY OF TEST RESULTS	3-1
3.3	TEST EXECUTION	3-2
APPENDIX A	MACRO PARAMETERS	
APPENDIX B	COMPILATION SYSTEM OPTIONS	
APPENDIX C	APPENDIX F OF THE Ada STANDARD	

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

INTRODUCTION

1.2 REFERENCES

- [Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8552-1987.
- [Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.
- [UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPRT13, and the procedure CHECK FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

INTRODUCTION

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

INTRODUCTION

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AV0. The rationale for withdrawing each test is available from either the AV0 or the AVF. The publication date for this list of withdrawn tests is 02 September 1990.

E28005C	B28006C	C34006D	B41308B	C43004A	C45114A
C45346A	C45612B	C45651A	C46022A	B49008A	A74006A
B83022B	B83022H	B83025B	B83025D	B83026A	C83026B
C83041A	B85001L	C97116A	C98003B	BA2011A	CB7001A
CB7001B	CB7004A	CC1223A	BC1226A	CC1226B	BC3009B
BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E	CD2A23E
CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C	BD3006A
CD4022A	CD4022D	CD4024B	CD4024C	CD4024D	CD4031A
CD4051D	CD5111A	CD7004C	ED7005D	CD7005E	AD7006A
CD7006E	AD7201A	AD7201E	CD7204B	BD8002A	BD8004C
CD9005A	CD9005B	CDA201E	CE2107I	CE2119B	CE2205B
CE2405A	CE3111C	CE3118A	CE3411B	CE3412B	CE3812A
CE3814A	CE3902B				

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

IMPLEMENTATION DEPENDENCIES

The following 201 tests have floating-point type declarations requiring more digits than `SYSTEM.MAX_DIGITS`:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

C35702A, C35713B, C45423B, B86001T, and C86006H check for the predefined type `SHORT_FLOAT`.

C35702B, C35713C, B86001U, and C86006G check for the predefined type `LONG_FLOAT`.

C35713D and B86001Z check for a predefined floating-point type with a name other than `FLOAT`, `LONG_FLOAT`, or `SHORT_FLOAT`.

A35801E checks that `FLOAT'FIRST..FLOAT'LAST` may be used as a range constraint in a floating-point type declaration; for this implementation that range exceeds the safe numbers and must be rejected. (See section 2.3)

C45423A, C45523A, and C45622A check that the proper exception is raised when operation results lie outside of the range of the base type if `MACHINE_OVERFLOW` is `TRUE` for various floating-point types; for this implementation, `MACHINE_OVERFLOW` is `FALSE`.

C45531M..P (4 tests) and C45532M..P (4 tests) check fixed-point operations for types that require a `SYSTEM.MAX_MANTISSA` of 47 or greater.

D64005G uses 17 levels of recursive procedure calls nesting which exceeds the capacity of the compiler.

C86001F recompiles package `SYSTEM`, making package `TEXT_IO`, and hence package `REPORT`, obsolete. For this implementation, the package `TEXT_IO` is dependent upon package `SYSTEM`.

B86001Y checks for a predefined fixed-point type other than `DURATION`.

CA2009C, CA2009F, BC3204C, and BC2205D instantiate generic units before their bodies are compiled. This implementation creates a dependence of generic units as allowed by AI-00408 and AI-00530 such that the compilation of the generic unit bodies makes the instantiating units obsolete. (See section 2.3)

LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests) check for pragma `INLINE` for procedures and functions.

IMPLEMENTATION DEPENDENCIES

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

AE2101C and EE2201D..E (2 tests) use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

The tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT FILE	SEQUENTIAL_IO
CE2102F	CREATE	INO UT FILE	DIRECT_IO
CE2102I	CREATE	IN FILE	DIRECT_IO
CE2102J	CREATE	OUT FILE	DIRECT_IO
CE2102N	OPEN	IN FILE	SEQUENTIAL_IO
CE2102O	RESET	IN FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT FILE	SEQUENTIAL_IO
CE2102R	OPEN	INO UT FILE	DIRECT_IO
CE2102S	RESET	INOUT FILE	DIRECT_IO
CE2102T	OPEN	IN FILE	DIRECT_IO
CE2102U	RESET	IN FILE	DIRECT_IO
CE2102V	OPEN	OUT FILE	DIRECT_IO
CE2102W	RESET	OUT FILE	DIRECT_IO
CE3102E	CREATE	IN FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	-----	TEXT_IO
CE3102I	CREATE	OUT FILE	TEXT_IO
CE3102J	OPEN	IN FILE	TEXT_IO
CE3102K	OPEN	OUT FILE	TEXT_IO

CE2107B..E (4 tests), CE2107L, CE2110B, and CE2111D attempt to associate multiple internal files with the same external file when one or more files is writing for sequential files.

IMPLEMENTATION DEPENDENCIES

CE2107G..H (2 tests), CE2110D, and CE2111H attempt to associate multiple internal files with the same external file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

CE2203A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for SEQUENTIAL_IO. This implementation does not restrict file capacity.

CE2403A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for DIRECT_IO. This implementation does not restrict file capacity.

CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A attempt to associate multiple internal files with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

CE3304A checks that USE_ERROR is raised if a call to SET_LINE_LENGTH or SET_PAGE_LENGTH specifies a value that is inappropriate for the external file. This implementation does not have inappropriate values for either line length or page length.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 12 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A B83033B B85013D

The following tests were split into multiple tests because of memory restrictions.

BC3205E AE2101A AE2101F

A35801E was graded inapplicable by Evaluation Modification as directed by the AVO; the compiler rejects the use of the range FLOAT'FIRST..FLOAT'LAST as the range constraint of a floating-point type declaration because the bounds lie outside of the range of safe numbers (cf. ARM 3.5.7(12)).

CA2009C, CA2009F, BC3204C, and BC3205D were graded inapplicable by Evaluation Modification as directed by the AVO. Because this implementation makes the units with instantiations obsolete (see section 2.2), the Class C tests were rejected at link time and the Class B tests were compiled without error.

EA1003B was processed with the option "-fI" so that code would be

IMPLEMENTATION DEPENDENCIES

generated for all of the legal units of this test file. Without this option, the entire compilation would have been rejected due to errors within only some of the units (which is also an acceptable result).

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for technical information about this Ada implementation system, see:

Technical Support
10 Pasteur Street
Irvine CA 92718
(714) 727-0700

For a point of contact for sales information about this Ada implementation system, see:

Jim Smith
10 Pasteur Street
Irvine CA 92718
(714) 727-0700

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

PROCESSING INFORMATION

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

Total Number of Applicable Tests	3807
Total Number of Withdrawn Tests	74
Processed Inapplicable Tests	88
Non-Processed I/O Tests	0
Non-Processed Floating-Point Precision Tests	201
Total Number of Inapplicable Tests	289
Total Number of Tests for ACVC 1.11	4170

All I/O tests of the test suite were processed because this implementation supports a file system. The above number of floating-point tests were not processed because they use floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 289 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation and 270 executable tests that use file operations not supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

Diskettes containing the customized test suite (see section 1.3) were taken on-site by the validation team for processing. The contents of the diskettes were transferred to the machine over a serial line from another PC machine (onto which the tests were directly loaded) using a program called LAPLINK.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix E for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

PROCESSING INFORMATION

Switch	Effect
-fE	Generate error file for the Ada listing utility (alu);
-fI	Ignore compilation errors and continue generating code for legal units within the same compilation file (for test EA1003B).
-fQ	Suppress "added to library" and "Generating code for" information messages.
-fV	Enable overflow checking (this is normally not specified directly by the user but is always provided by the compilation system).
-fw	Suppress informative warning messages.
-lc	Produce continuous form Ada listing (no page headers).
-lp	Obey PRAGMA PAGE directives within the program even though the -c flag says not to generate page breaks.
-ls	Output Ada listing to the standard output file instead of to a disk file.

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for \$MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
\$BIG_ID1	(1..V-1 => 'A', V => '1')
\$BIG_ID2	(1..V-1 => 'A', V => '2')
\$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
\$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
\$BIG_INT_LIT	(1..V-3 => '0') & "298"
\$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
\$BIG_STRING1	"" & (1..V/2 => 'A') & ""
\$BIG_STRING2	"" & (1..V-1-V/2 => 'A') & '1' & ""
\$BLANKS	(1..V-20 => ' ')
\$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
\$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
\$MAX_STRING_LITERAL	"" & (1..V-2 => 'A') & ""

MACRO PARAMETERS

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
\$MAX_IN_LEN	200
\$ACC_SIZE	32
\$ALIGNMENT	2
\$COUNT_LAST	32766
\$DEFAULT_MEM_SIZE	1024
\$DEFAULT_STOR_UNIT	8
\$DEFAULT_SYS_NAME	I8086
\$DELTA_DOC	2.0**(-31)
\$ENTRY_ADDRESS	16#0#
\$ENTRY_ADDRESS1	16#1#
\$ENTRY_ADDRESS2	16#2#
\$FIELD_LAST	32767
\$FILE_TERMINATOR	' '
\$FIXED_NAME	NO_SUCH_FIXED_TYPE
\$FLOAT_NAME	NO_SUCH_FLOAT_TYPE
\$FORM_STRING	" "
\$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
\$GREATER_THAN_DURATION	90000.0
\$GREATER_THAN_DURATION_BASE_LAST	10000000.0
\$GREATER_THAN_FLOAT_BASE_LAST	1.8E+308
\$GREATER_THAN_FLOAT_SAFE_LARGE	1.0E308

MACRO PARAMETERS

```

$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE
    1.0E308

$HIGH_PRIORITY        20

$ILLEGAL_EXTERNAL_FILE_NAME1
    \NODIRECTORY\FILENAME1

$ILLEGAL_EXTERNAL_FILE_NAME2
    \NODIRECTORY\FILENAME2

$INAPPROPRIATE_LINE_LENGTH
    -1

$INAPPROPRIATE_PAGE_LENGTH
    -1

$INCLUDE_PRAGMA1      PRAGMA INCLUDE ("A28006D1.ADA")

$INCLUDE_PRAGMA2      PRAGMA INCLUDE ("B28006F1.ADA")

$INTEGER_FIRST        -32768

$INTEGER_LAST         32767

$INTEGER_LAST_PLUS_1  32768

$INTERFACE_LANGUAGE   C

$LESS_THAN_DURATION   -90000.0

$LESS_THAN_DURATION_BASE_FIRST
    -10000000.0

$LINE_TERMINATOR      ASCII.CR & ASCII.LF

$LOW_PRIORITY         1

$MACHINE_CODE_STATEMENT
    INST1'(B1=>16#90#);

$MACHINE_CODE_TYPE     INST1

$MANTISSA_DOC          31

$MAX_DIGITS           15

$MAX_INT              2147483647

$MAX_INT_PLUS_1       2147483648

$MIN_INT              -2147483648

```

MACRO PARAMETERS

\$NAME	BYTE_INTEGER
\$NAME_LIST	I8086
\$NAME_SPECIFICATION1	C:\CEA\X2120A
\$NAME_SPECIFICATION2	C:\CEA\X2120B
\$NAME_SPECIFICATION3	C:\CEB\X3119A
\$NEG_BASED_INT	16#FFFFFFFFE#
\$NEW_MEM_SIZE	1024
\$NEW_STOR_UNIT	8
\$NEW_SYS_NAME	I8086
\$PAGE_TERMINATOR	ASCII.CR & ASCII.LF & ASCII.FF
\$RECORD_DEFINITION	RECORD B1: UNSIGNED_BYTE; END RECORD;
\$RECORD_NAME	INST1
\$TASK_SIZE	32
\$TASK_STORAGE_SIZE	2048
\$TICK	1.0/18.2
\$VARIABLE_ADDRESS	FCNDECL.VAR_ADDRESS
\$VARIABLE_ADDRESS1	FCNDECL.VAR_ADDRESS1
\$VARIABLE_ADDRESS2	FCNDECL.VAR_ADDRESS2
\$YOUR_PRAGMA	NO_SUCH_PRAGMA

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

MERIDIAN Ada COMPILER OPTIONS

- fD Generate debugging output. The -fD option causes the compiler to generate the appropriate code and data for operation with the Meridian Ada Debugger.
- fe Annotate assembly language listing. The -fe option causes the compiler to annotate an assembly language output file. The output is supplemented by comments containing the Ada source statements corresponding to the assembly language code sections written by the code generator. To use this option, the -S option must also be specified; otherwise, the annotated file is not emitted.
- fE Generate error log file. The -fE option causes the compiler to generate a log file containing all the error messages and warning messages produced during compilation. The error log file has the same name as the source file, with the extension .err. For example, the error log file for simple.adb is simple.err. The error log file is placed in the current working directory. In the absence of the -fE option, the error log information is sent to the standard output stream.
- fF Disable floating point checks. This option is used to inhibit checks for a math co-processor before sequences of math co-processor instructions, resulting in a slightly smaller and faster program. Use of this option means that the resulting program requires, and you guarantee, the run-time presence of a math co-processor.

COMPILATION SYSTEM OPTIONS

(either an 8087, 80287, or 80387). If a program containing floating point computations is compiled with the -fF option, it will behave unpredictably if run on a machine without a math co-processor installed; the machine may simply "freeze up" in this circumstance, requiring a reboot. Refer to the bamp -u option, which causes the floating point software to be linked with a program.

- fI Ignore compilation errors and continue generating code for legal units within the same compilation file.
- fL Generate exception location information. The -fL option causes location information (source file names and line numbers) to be maintained for internal checks. This information is useful for debugging in the event that an "Exception never handled" message appears when an exception propagates out of the main program. This flag causes the code to be somewhat larger. If -fL is not used, exceptions that propagate out of the main program will behave in the same way, but no location information will be printed with the "Exception never handled" message.
- fN Suppress numeric checking. The -fN flag suppresses two kinds of numeric checks for the entire compilation: division check and overflow check. These checks are described in section 11.7 of the LRM. This flag reduces the size of the code.
- fQ Suppress "added to library" and "Generating code for" information messages normally output by the compiler.
- fR Inhibit static initialization of variables. This option is intended for use in ROM-based embedded environments in conjunction with the Meridian Ada Run-Time Customization Library. The -fR option is applicable only in the presence of the -fs option, which suppresses certain run time checks. Normally, the Ada compiler initializes constants or variables with static data when the following conditions all occur:
 1. Checking is disabled with the -fs option.
 2. The initializer expression is static (known at compile time).
 3. The object is a global (in top-level package specification or body).

If the -fR flag is specified, static initialization is

COMPILATION SYSTEM OPTIONS

suppressed for variables (but not for constants); assignments to each component of a variable are performed in the code. Note that this always happens in the absence of the -fs option.

- fs Suppress all checks. The -fs flag suppresses all automatic checking, including numeric checking. This flag is equivalent to using pragma suppress on all checks. This flag reduces the size of the code, and is good for producing "production quality" code or for benchmarking the compiler. Note that there is a related ada option, -fN to suppress only certain kinds of numeric checks.
- fS The -fS flag causes the compiler to generate additional 80286 instructions not available on the 8086/8088. Programs compiled in this mode tend to be smaller than programs compiled using the normal 8086/8088 mode.
- fU Inhibit library update. The -fU option inhibits library updates. This is of use in conjunction with the -S option. Certain restrictions apply to use of this option.
- fv Compile verbosely. The compiler prints the name of each subprogram, package, or generic as it is compiled. Information about the symbol table space remaining following compilation of the named entity is also printed in the form "[nK]".
- fV Enable overflow checking (this is normally not specified directly by the user but is always provided by the compilation system).
- fw Suppress warning messages. With this option, the compiler does not print warning messages about ignored pragmas, exceptions that are certain to be raised at run-time, or other potential problems that the compiler is otherwise forbidden to deem as errors by the LRM.
- g The -g option instructs the compiler to run an additional optimization pass. The optimizer removes common sub-expressions, dead code and unnecessary jumps. It also does loop optimizations.
- K Keep internal form file. This option is used in conjunction with the Optimizer. Without this option, the compiler deletes internal form files following code generation.

COMPILATION SYSTEM OPTIONS

-lmodifiers

Generate listing file. The -l option causes the compiler to create a listing. Optional modifiers can be given to affect the listing format. You can use none or any combination of the following modifiers:

- c Use continuous listing format. The listing by default contains a header on each page. Specifying -lc suppresses both pagination and header output, producing a continuous listing.
- p Obey pragma page directives. Specifying -lp is only meaningful if -lc has also been given. Normally -lc suppresses all pagination, whereas -lcp suppresses all pagination except where explicitly called for within the source file with a pragma page directive.
- s Use standard output. The listing by default is written to a file with the same name as the source file and the extension .lst, as in simple.lst from simple.adb. Specifying -ls causes the listing file to be written to the standard output stream instead.
- t Generate relevant text output only. The listing by default contains the entire source program as well as interspersed error messages and warning messages. Specifying -lt causes the compiler to list only the source lines to which error messages or warning messages apply, followed by the messages themselves.

The default listing file generated has the same name as the source file, with the extension .lst. For example, the default listing file produced for simple.adb has the name simple.lst. The listing file is placed in the current working directory. Note: -l also causes an error log file to be produced, as with the -fE option.

-L library-name

Default: ada.lib

Use alternate library. The -L option specifies an alternative name for the program library.

- N No compile. This option causes the ada command to do a "dry run" of the compilation process. The command invoked for each processing step is printed. This is similar to the -P option, but no actual processing is performed.
- P Print compile. This option causes the ada command to

COMPILATION SYSTEM OPTIONS

print out the command invoked for each processing step as it is performed.

- S Produce assembly code. This option causes the code generator to produce an assembly language source file and to halt further processing.

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

MERIDIAN Ada LINKER OPTIONS

- A Aggressively inline. This option instructs the optimizer to aggressively inline subprograms when used in addition to the -G option. Typically, this means that subprograms that are only called once are inlined. If only the -G option is used, only subprograms for which pragma inline has been specified are inlined.

- c compiler-program-name

Default: (as stored in program library)

Use alternate compiler. The -c option specifies the complete (non-relative) directory path to the Meridian Ada compiler. This option overrides the compiler program name stored in the program library. The -c option is intended for use in cross-compiler configurations, although under such circumstances, an appropriate library configuration is normally used instead.

- f Suppress main program generation step. The -f option suppresses the creation and additional code generation steps for the temporary main program file. The -f option can be used when a simple change has been made to the body of a compilation unit. If unit elaboration order is changed, if the specification of a unit is changed, or if new units are added, then this option should not be used.
- g Perform global optimization only. The -g option causes bamp to invoke the global optimizer on your program.

COMPILATION SYSTEM OPTIONS

Compilation units to be optimized globally must have been compiled with the ada -K option.

- G Perform global and local optimization. The -G option causes bamp to perform both global and local optimization on your program. This includes performing pragma inline. As with the -g option, compilation units to be optimized must have been compiled with the ada -K option.
- i The -i option is used in conjunction with the bamp -r option when producing "pre-linked" code for use with the Intel Development Tools. The -i option causes certain information to be emitted into the object file that is needed under some circumstances by the Intel linker, LINK86. By default, pre-linked object modules use the Microsoft object format.
- I Link the program with a version of the tasking run-time which supports pre-emptive task scheduling. This option produces code which handles interrupts more quickly, but has a slight negative impact on performance in general.

-L library-name

Default: ada.lib

Use alternate library. The -L option specifies the name of the program library to be consulted by the bamp program. This option overrides the default library name.

- m Produce link map. The -m option causes a text file containing a link map to be written. The link map is Microsoft-compatible and the link map file name has the extension .map for Real Mode programs (the default). For Extended Mode Programs (produced when the bamp -x option is given), the link map is OS/x86-compatible and the link map file name has the extension .xmp.

-M main-program-stack-size

Default:

- o 20K in Real Mode programs
- o 64K in Extended Mode programs. when tasking is not used
- o 64K - <task-stack-size> in Extended Mode programs, when tasking is used

Set main program stack size. The -M option sets the stack size (number of decimal bytes) for the main program

COMPILATION SYSTEM OPTIONS

(excluding tasking). Note that the sum of the main program stack size and the tasking stack size must be no more than 64K bytes.

- n No link. The -n option suppresses actual object file linkage, but creates and performs code generation on the main program file.
- N No operations. The -N option causes the bamp command to do a "dry run"; it prints out the actions it takes to generate the executable program, but does not actually perform those actions. The same kind of information is printed by the -P option.

-o output-file-name

Default: file.exe

Use alternate executable file output name. The -o option specifies the name of the executable program file written by the bamp command. This option overrides the default output file name.

- P Print operations. The -P option causes the bamp command to print out the actions it takes to generate the executable program as the actions are performed.
- r Create re-linkable output. The -r option causes an object file (.obj file) to be generated rather than an executable file (.exe file). The resulting file contains all symbol and relocation information, and can then be used with any low-level linker accepting object files compatible with the Intel or Microsoft object formats.

-s task-stack-size

Default:

- o 20K if tasking used
- o Zero if tasking not used

Use alternate tasking stack size. The -s option specifies the number of bytes (in decimal) to be allocated to all the tasks to be activated in the Ada program. This option overrides the default task stack size. Note that the sum of the main program stack size and the tasking stack size must be somewhat smaller than 64K bytes. The size of individual task activation stacks can be specified with a length clause.

- u Link software floating point library. Use of the -u

COMPILATION SYSTEM OPTIONS

option enables a program containing floating point computations to run with or without a math co-processor chip. A related compiler option, the ada -fF option, also can be used to control the action of the run-time in the absence of a math co-processor chip. The ada -fF option and the bamp -u option should not both be used in the same program.

- v Link verbosely. The -v option causes the bamp command to print out information about what actions it takes in building the main program.
- V scratch-file
Link using "virtual" mode. This option allows larger programs to be linked, although slightly more slowly. A scratch-file must be specified. The scratch-file can reside on a RAM disk (if one is available) for faster operation. The -V option affects only the operation of the low-level object linker. The scratch-file is used as scratch memory in which the various object files are linked.
- W Suppress warnings. This option allows you to suppress warnings from the optimizer.
- x The -x option is used to create an Extended Mode program. This option applies only to Extended Mode Meridian Ada. The -x option produces a program that can be run with the ramp command to run in Extended Mode (a .exp file). If the -x option is not used, a Real Mode program (a .exe file) is produced.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

package STANDARD is

...

type INTEGER is range -32768 .. 32767;

type FLOAT is digits 15

range -1.79769313486231E+308 .. 1.79769313486231E+308;

type DURATION is delta 0.0001 range -86400.0 .. 86400.0;

type LONG_INTEGER is range -2147483648 .. 2147483647;

type SHORT_INTEGER is range -32768 .. 32767;

type BYTE_INTEGER is range -128 .. 127;

...

end STANDARD;

Appendix F Implementation-Dependent Characteristics

This appendix lists implementation-dependent characteristics of Meridian Ada. Note that there are no preceding appendices. This appendix is called *Appendix F* in order to comply with the Reference Manual for the Ada Programming Language* (LRM) ANSI/MIL-STD-1815A which states that this appendix be named Appendix F.

Implemented Chapter 13 features include length clauses, enumeration representation clauses, record representation clauses, address clauses, interrupts, package **system**, machine code insertions, pragma **interface**, and unchecked programming.

F.1 Pragmas

The implemented pre-defined pragmas are:

elaborate	See the LRM section 10.5.
interface	See section F.1.1.
list	See the LRM Appendix B.
pack	See section F.1.2.
page	See the LRM Appendix B.
priority	See the LRM Appendix B.
suppress	See section F.1.3.
inline	See the LRM section 6.3.2. This pragma is not actually effective unless you compile/link your program using the global optimizer.

The remaining pre-defined pragmas are accepted, but presently ignored:

controlled	optimize	system_name
shared	storage_unit	
memory_size		

Named parameter notation for pragmas is not supported.

When illegal parameter forms are encountered at compile time, the compiler issues a warning message rather than an error, as required by the Ada language definition. Refer to the LRM Appendix B for additional information about the pre-defined pragmas.

F.1.1 Pragma Interface

The form of pragma **interface** in Meridian Ada is:

```
pragma interface ( language, subprogram [, "link-name" ] );
```

where:

<i>language</i>	is the interface language, one of the names assembly , builtin , c , microsoft_c , or internal . The names builtin and internal are reserved for use by Meridian compiler maintainers in run-time support packages.
<i>subprogram</i>	is the name of a subprogram to which the pragma interface applies.
<i>link-name</i>	is an optional string literal specifying the name of the non-Ada subprogram corresponding to the Ada subprogram named in the second parameter. If <i>link-name</i> is omitted, then <i>link-</i>

*All future references to the Reference Manual for the Ada Programming Language appear as the LRM.

name defaults to the value of *subprogram*. Depending on the language specified, some automatic modifications may be made to the *link-name* to produce the actual object code symbol name that is generated whenever references are made to the corresponding Ada subprogram. The object code symbol generated for *link-name* is always translated to upper case. Although the Meridian object linker is case-sensitive, it is a rare object module that contains mixed-case symbols; at present, all Meridian 80x86 object modules use upper case only.

It is appropriate to use the optional *link-name* parameter to pragma **interface** only when the interface subprogram has a name that does not correspond at all to its Ada identifier or when the interface subprogram name cannot be given using rules for constructing Ada identifiers (e.g. if the name contains a '\$' character).

The characteristics of object code symbols generated for each interface language are:

assembly	The object code symbol is the same as <i>link-name</i> .
builtin	The object code symbol is the same as <i>link-name</i> , but prefixed with two underscore characters ("_ _"). This language interface is reserved for special interfaces defined by Meridian Software Systems, Inc. The builtin interface is presently used to declare certain low-level run-time operations whose names must not conflict with programmer-defined or language system defined names.
c	The object code symbol is the same as <i>link-name</i> , but with one underscore character ('_') prepended. This is the convention used by the C compiler.
internal	No object code symbol is generated for an internal language interface; this language interface is reserved for special interfaces defined by Meridian Software Systems, Inc. The internal interface is presently used to declare certain machine-level bit operations.
microsoft_c	The object code symbol is the same as <i>link-name</i> , but with one underscore character ('_') prepended. This is the convention used by the Microsoft C compiler.

The low-level calling conventions are changed only in the case of a **microsoft_c** interface. No automatic data conversions are performed on parameters of any interface subprograms. It is up to the programmer to ensure that calling conventions match and that any necessary data conversions take place when calling interface subprograms.

A pragma **interface** may appear within the same declarative part as the subprogram to which the pragma **interface** applies, following the subprogram declaration, and prior to the first use of the subprogram. A pragma **interface** that applies to a subprogram declared in a package specification must occur within the same package specification as the subprogram declaration; the pragma **interface** may not appear in the package body in this case. A pragma **interface** declaration for either a private or nonprivate subprogram declaration may appear in the private part of a package specification.

Pragma **interface** for library units is not supported.

Refer to the LRM section 13.9 for additional information about pragma **interface**.

F.1.2 Pragma Pack

Pragma **pack** is implemented for composite types (records and arrays).

Pragma **pack** is permitted following the composite type declaration to which it applies, provided that the pragma occurs within the same declarative part as the composite type declaration, before any objects or components of the composite type are declared.

Note that the declarative part restriction means that the type declaration and accompanying pragma **pack** cannot be split across a package specification and body.

The effect of `pragma pack` is to minimize storage consumption by discrete component types whose ranges permit packing. Use of `pragma pack` does not defeat allocations of alignment storage gaps for some record types. `Pragma pack` does not affect the representations of real types, pre-defined integer types, and access types.

F.1.3 Pragma Suppress

`Pragma suppress` is implemented as described in the LRM section 11.7, with these differences:

- Presently, `division_check` and `overflow_check` must be suppressed via a compiler flag, `-tN`; `pragma suppress` is ignored for these two numeric checks.
- The optional "ON =>" parameter name notation for `pragma suppress` is ignored.
- The optional second parameter to `pragma suppress` is ignored; the pragma always applies to the entire scope in which it appears.

F.2 Attributes

All attributes described in the LRM Appendix A are supported. The implementation-dependent Meridian attribute `'locoffset` is applied to a parameter and returns as a universal-integer the stack offset of that parameter (the offset from the BP register). It allows machine code insertions to access parameters using less error-prone symbolic names. An example follows.

```
machine_code.inst3'(16#8B#, 16#4E#, nbytes'locoffset);
```

F.3 Standard Types

Additional standard types are defined in Meridian Ada:

- `byte_integer`
- `short_integer`
- `long_integer`

The standard numeric types are defined as:

```
type byte_integer is range -128 .. 127;
type short_integer is range -32768 .. 32767;
type integer is range -32768 .. 32767;
type long_integer is range -2147483648 .. 2147483647;
type float is digits 15
  range -1.79769313486231E+308 .. 1.79769313486231E+308;
type duration is delta 0.0001 range -86400.0000 .. 86400.0000;
```

F.4 Package System

The specification of package `system` is:

```
package system is
  type address is new long_integer;
  type name is (i8086);
  system_name : constant name := i8086;
```

```

storage_unit  : constant := 8;
memory_size   : constant := 1024;

-- System-Dependent Named Numbers
min_int       : constant := -2147483648;
max_int       : constant := 2147483647;
max_digits    : constant := 15;
max_mantissa  : constant := 31;
fine_delta    : constant := 2.0 ** (-31);
tick          : constant := 1.0 / 18.2;

-- Other System-Dependent Declarations
subtype priority is integer range 1 .. 20;

```

The value of `system.memory_size` is presently meaningless.

F.5 Restrictions on Representation Clauses

F.5.1 Length Clauses

A size specification (`t'size`) is rejected if fewer bits are specified than can accommodate the type. The minimum size of a composite type may be subject to application of pragma `pack`. It is permitted to specify precise sizes for unsigned integer ranges, e.g. 8 for the range `0 .. 255`. However, because of requirements imposed by the Ada language definition, a full 32-bit range of unsigned values, i.e. `0 .. (2**32) - 1`, cannot be defined, even using a size specification.

The specification of collection size (`t'storage_size`) is evaluated at run-time when the scope of the type to which the length clause applies is entered, and is therefore subject to rejection (via `storage_error`) based on available storage at the time the allocation is made. A collection may include storage used for run-time administration of the collection, and therefore should not be expected to accommodate a specific number of objects. Furthermore, certain classes of objects such as unconstrained discriminant array components of records may be allocated outside a given collection, so a collection may accommodate more objects than might be expected.

The specification of storage for a task activation (`t'storage_size`) is evaluated at run-time when a task to which the length clause applies is activated, and is therefore subject to rejection (via `storage_error`) based on available storage at the time the allocation is made. Storage reserved for a task activation is separate from storage needed for any collections defined within a task body.

The specification of `small` for a fixed point type (`t'small`) is subject only to restrictions defined in the LRM section 13.2.

F.5.2 Enumeration Representation Clauses

The internal code for the literal of an enumeration type named in an enumeration representation clause must be in the range of `standard.integer`.

The value of an internal code may be obtained by applying an appropriate instantiation of `unchecked_conversion` to an integer type.

F.5.3 Record Representation Clauses

The storage unit offset (the `at static_simple_expression` part) is given in terms of 8-bit storage units and must be even.

A bit position (the *range* part) applied to a discrete type component may be in the range 0 . . 15, with 0 being the least significant bit of a component. A *range* specification may not specify a size smaller than can accommodate the component. A *range* specification for a component not accommodating bit packing may have a higher upper bound as appropriate (e.g. 0 . . 31 for a discriminant **string** component). Refer to the internal data representation of a given component in determining the component size and assigning offsets.

Components of discrete types for which bit positions are specified may not straddle 16-bit word boundaries.

The value of an alignment clause (the optional **at mod** part) must evaluate to 1, 2, 4, or 8, and may not be smaller than the highest alignment required by any component of the record. On PC-DOS, this means that some records may not have alignment clauses smaller than 2.

F.5.4 Address Clauses

An address clause may be supplied for an object (whether constant or variable) or a task entry, but not for a subprogram, package, or task unit. The meaning of an address clause supplied for a task entry is given in section F.5.5.

An address expression for an object is a 32-bit segmented memory address of type **system.address**.

F.5.5 Interrupts

A task entry's address clause can be used to associate the entry with a PC-DOS interrupt. Values in the range 0 . . 255 are meaningful, and represent the interrupts corresponding to those values.

An interrupt entry may not have any parameters.

F.5.6 Change of Representation

There are no restrictions for changes of representation effected by means of type conversion.

F.6 Implementation-Dependent Components

No names are generated by the implementation to denote implementation-dependent components.

F.7 Unchecked Conversions

There are no restrictions on the use of **unchecked_conversion**. Conversions between objects whose sizes do not conform may result in storage areas with undefined values.

F.8 Input-Output Packages

A summary of the implementation-dependent input-output characteristics is:

- In calls to **open** and **create**, the *form* parameter must be the empty string (the default value).
- More than one internal file can be associated with a single external file for reading only. For writing, only one internal file may be associated with an external file; Do not use **reset** to get around this rule.
- Temporary sequential and direct files are given names. Temporary files are deleted when they are closed.
- File I/O is buffered; text files associated with terminal devices are line-buffered.
- The packages **sequential_io** and **direct_io** cannot be instantiated with unconstrained composite types or record types with discriminants without defaults.

Appendix F

F.9 Source Line and Identifier Lengths

Source lines and identifiers in Ada source programs are presently limited to 200 characters in length.